IRIDIS 1™

Four Excellent Programs for Your ATARI® 800

☐ **CLOCK**

No—this isn't another digital clock! It's an old-fashioned wall clock, with chimes to charm you. And it keeps on ticking . . .

☐ **ZAP**

Grab a joystick and try to zap the targets. Has an "attract" mode when you don't want to play. You'll also learn how to use the ATARI START button in your Basic programs. *(Needs one joystick.)*

☐ **LOGO**

A flashy demo that shows off the ATARI color registers. Comes with a subroutine that makes drawing pictures easier.

☐ **POLYGONS**

Your ATARI constructs beautiful geometric patterns.

16k memory needed when using IRIDIS programs with cassette, and 24k needed with the ATARI 810 disk.

Published By:

The Code Works™

Box 550
Goleta, CA 93017

Comments

IRIDIS 1 is the first in a series of tutorials about the Atari personal computer. Each one brings you four excellent programs on cassette (or disk), ready to Run. You also receive this printed Guide, which explains how to use the IRIDIS programs, as well as the technical background about how they work, including complete source code listings of selected programs.

There are two regular columns that we think you'll enjoy: "Novice Notes" for the person that is brand new to computing, and "Hacker's Delight" for the more advanced bit twiddlers. We think that both beginners and old hands will find that studying IRIDIS programs is one of the best ways to learn more about the Atari.

Please note that the programs in IRIDIS 1 require 16k of memory if you are using the Atari 410 program recorder, or 24k when using the Atari 810 disk.

Glen Fisher and Ron Jeffries
The Code Works

Clock

The CLOCK program draws an old-fashioned clock (the kind with hands, and not little red numbers), and starts ticking off the seconds. When you run the program, it first waits for you to press [RETURN]. After you press [RETURN], there will be a short delay while it sets up its time-keeping routine. Next, it asks you to enter the time. Type the hour, minute, and second, separated by commas. CLOCK accepts the time in either the normal twelve-hour system, or the military-style twenty-four-hour time. (In either case, it only displays twelve-hour time.)

CLOCK chimes the hours, just like your favorite grandfather clock. It also sounds the quarter hours, with one tone for each quarter-hour past the hour: one for quarter past, two for the half hour, and three for quarter till the hour.

Note: since the CLOCK program insinuates itself into the Atari operating system (to keep as accurate time as possible), you should use [RESET] to stop the program. That will cause the Atari to kick CLOCK out of the system again. Otherwise, the I/O won't necessarily work.

```
0 REM CLOCK
1 REM COPYRIGHT(C) 1980 IRIDIS
2 REM BOX 550, GOLETA, CA. 93017
3 REM ALL RIGHTS RESERVED
10 REM AS OF 4 MAR 80
90 GOSUB 30000
100 PRINT "{CLEAR}"
110 DIM VO(59),HO(59),VI(59),HI(59)
120 DEG :CIRCLE=360
180 HCENT=160:VCENT=96:LARGE=90:SMALL=60
200 GOSUB 4000
470 PRINT "{CLEAR 3 DOWN}":GOTO 490
480 PRINT "I don't understand that."
490 TRAP 480
500 PRINT :PRINT "Please enter the time:"
510 PRINT "(as HR,MIN,SEC): ";
520 INPUT HR,MIN,SEC
530 TRAP 65535
550 IF HR>12 THEN HR=HR-12
560 IF HR<1 OR HR>12 THEN PRINT "Hour must be from 1 to 12.":GOTO 490
570 IF MIN<0 OR MIN>59 THEN PRINT "Minutes must be from 0 to 59.":GOTO 490
580 IF SEC<0 OR SEC>59 THEN PRINT "Seconds must be from 0 to 59.":GOTO 490
680 N=INT(MIN/12):FIFTHHR=MIN-12*N:HR=HR*5+N
690 POKE 204,SEC:POKE 205,MIN:POKE 206,FIFTHHR:POKE 207,HR
700 GRAPHICS 24
710 SETCOLOR 4,8,0
720 SETCOLOR 2,8,0
730 SETCOLOR 1,8,14
740 COLOR 1
750 N=60:J=4
760 FOR I=0 TO N-1
770 V=-COS(CIRCLE*I/N)
780 H=SIN(CIRCLE*I/N)
790 HO(I)=INT(LARGE*H+HCENT)
800 VO(I)=INT(LARGE*V+VCENT)
```

```
810 PLOT INT(H*(LARGE+5)+HCENT),INT(V*(LARGE+5)+VCENT)
820 J=J+1:IF J=5 THEN DRAWTO INT(H*(LARGE+2)+HCENT),INT(V*(LARGE+2))+VCENT:J=0
830 HI(I)=INT(SMALL*H+HCENT)
840 VI(I)=INT(SMALL*V+VCENT)
850 NEXT I
900 SEC=PEEK(204):MIN=PEEK(205):HR=PEEK(207)
910 GOTO 1200
1000 SEC=PEEK(204):MIN=PEEK(205):HR=PEEK(207)
1010 COLOR 0
1100 IF OLDSEC<>LASTMIN THEN PLOT HI(OLDSEC),VI(OLDSEC):DRAWTO HO(OLDSEC),VO(OLDSEC)
1110 IF OLDMIN=MIN THEN 1300
1120 PLOT HCENT,VCENT:DRAWTO HO(OLDMIN),VO(OLDMIN):OLDMIN=MIN
1125 I=MIN/15:IF I=INT(I) THEN QUARTER=I
1130 IF OLDHR<>HR AND MIN=0 THEN CHIME=HR/5:IF CHIME=0 THEN CHIME=12
1140 IF OLDHR<>HR THEN PLOT HCENT,VCENT:DRAWTO HI(OLDHR),VI(OLDHR):OLDHR=HR
1200 COLOR 1
1210 PLOT HCENT,VCENT:DRAWTO HI(HR),VI(HR)
1220 PLOT HCENT,VCENT:DRAWTO HO(MIN),VO(MIN)
1300 COLOR 1
1310 IF SEC<>MIN THEN PLOT HI(SEC),VI(SEC):DRAWTO HO(SEC),VO(SEC)
1320 OLDSEC=SEC:LASTMIN=MIN
1330 SOUND 0,0,4,14:I=I+1:SOUND 0,0,0,0
1410 IF CHIME>0 THEN VOL=8-VOL:SOUND 1,255,10,VOL:SOUND 2,252,10,VOL:
     IF VOL=0 THEN CHIME=CHIME-1
1420 IF QUARTER>0 THEN VOL=8-VOL:SOUND 1,63,10,VOL:
     IF VOL=0 THEN QUARTER=QUARTER-1
1800 IF PEEK(204)=SEC THEN 1800
1810 GOTO 1000
3000 DATA A2,00,E6,CB,A5,CB,C9,3C,90,2A
3010 DATA 86,CB,E6,CC,A5,CC,C9,3C
3020 DATA 90,20,86,CC,E6,CD,A5,CD
3030 DATA C9,3C,90,02,86,CD,E6,CE
3040 DATA A5,CE,C9,0C,90,0C,86,CE
3050 DATA E6,CF,A5,CF,C9,3C,90,02
3060 DATA 86,CF,4C,62,E4,==
4000 DIM HX$(2)
4005 PRINT "Setting up my timer..."
4010 J=960:RESTORE 3000
4020 READ HX$:IF HX$="==" THEN 4100
4030 H=ASC(HX$(1,1))-48:IF H>9 THEN H=H-7
4040 L=ASC(HX$(2,2))-48:IF L>9 THEN L=L-7
4050 POKE J,H*16+L:J=J+1:GOTO 4020
4100 POKE 54286,0:POKE 548,192:POKE 549,3:POKE 54286,64
4200 RETURN
30000 DIM CR$(1):CR$=CHR$(155)
30010 GRAPHICS 2:OPEN #1,4,0,"K:":POKE 752,1
30020 SETCOLOR 0,8,12:SETCOLOR 3,9,4:SETCOLOR 2,0,0
30030 PRINT #6;CR$;CR$;CR$;CR$;
30040 PRINT #6;"      {9 C}"
30050 PRINT #6;"      {C} CLOCK {C}"
30060 PRINT #6;"      {9 C}"
30070 PRINT "{DOWN}     COPYRIGHT (C) 1980  IRIDIS"
30080 PRINT "{DOWN}       PRESS RETURN TO BEGIN.";
30090 GET #1,T:CLOSE #1:CLR :POKE 752,0:GOTO 100
```

Behind the Scenes

To maintain the clock picture, CLOCK needs to remember two sets of positions: those of the end of the hour hand, and of the end of the minute hand. Almost all of the code in CLOCK is concerned with keeping track of the hand positions, and keeping the picture intact.

===== Arrays =====

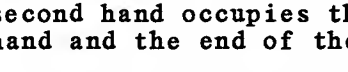
HO() Horizontal positions in Outer circle: ends of the minute hand

VO() Vertical positions in Outer circle: ditto

HI() Horizontal positions in Inner circle: the ends of the hour hand

VI() Vertical positions in Inner circle: ditto

The second hand occupies the area between the end of the hour hand and the end of the minute hand.



===== Variables =====

HR the position of the hour hand (NOT the hour of the day!)

MIN the position of the minute hand

SEC the position of the second hand

FIFTHHR counts fifths-of-hours. Every fifth of an hour (twelve minutes), the hour hand is moved one position. (This variable is used only in setting up the time.)

OLDSEC value of SEC before the second changed

LASTMIN The value of MIN one second ago.

OLDMIN value of MIN before the minute changed

OLDHR value of HR before the 'hour' changed

CHIME how many times the hour chime should sound

QUARTER how many times the quarter-hour chime should sound

VOL volume of the chimes (alternates between 0 and 8)

Other variables (any not named above) are general use temporary variables.

===== Constants =====

CIRCLE the circumference of a circle, in degrees

HCENT horizontal position of the center of the clock

VCENT vertical position of the center

LARGE length of the minute hand

SMALL length of the hour hand

===== The program =====

100 Clear the screen (actually just the text window).

110 Create the arrays.

128-180 Make Basic work in degrees, and set up the constants.

200 Call the subroutine to set up the timer

470 Clear the text window, in preparation for getting the time. Skip around the illegal-input message. (Clearing the text window is needed despite line 100, because the subroutine called from line 200 prints a message.)
480 Say that he typed in the time wrong, but politely.
490 If he blows the input, go to line 480, and say something nasty.
500-520 Ask for the time.
530 Don't intercept errors any more.
550 If it's twenty-four-hour time, convert it to twelve-hour time.
560-580 Make sure the time is a real one (10:70:43 isn't real).
680 Figure out which fifth of the hour it is, and where the hour hand should be. (The minute and seconds hand positions are already known.)
690 Tell the machine-language timer what time it is.
700 Enter full screen, highest resolution graphics mode.
710-740 Make the clock light blue on a dark blue background.
750 N=number of dots around the clock (done for convenience). J=counter for placement of the hour marks.
760 For each dot around the edge of the clock...
770-780 Find the vertical (V) and horizontal (H) positions of the minute dots on a unit circle (a circle whose radius is one unit).
790-800 Convert the position to that of the end of the minute hand and remember the converted position.
810 Put the minute dot on the screen (out from the end of the minute hand).
820 If this is the fifth dot since the last hour mark, put up a new hour mark.
830-840 Convert the dot position to the end of the hour hand, and remember the position.
850 End of loop
900 Find out what time it is now (drawing all those dots took time!).
910 Go off and display the hands.

===== Main loop, keeping the clock picture on time.
1000 Find out what time it is.
1010 Set up to erase clock hands
1100 If the second hand isn't under the minute hand, erase it.
1110 If the minute hand shouldn't move, skip to hand-drawing code.
1120 Erase old minute hand (and possible part of the hour hand).
1125 See if we're at a quarter hour, and arrange for a chime if so.
1130 See if we're at a full hour, and arrange for a different chime if so.

1140 If the hour hand should move, erase it.
1150 Draw the new minute and hour hands. We must draw the hour hand even if it didn't move, because erasing the minute hand may have erased part of the hour hand.
1300 Prepare to draw the new second hand.
1310 If the the second hand isn't under the minute hand, draw it.
1320 Remember where the minute and second hands are right now, so we can erase the right things next time around.
1330 Do the tick. I=I+1 is there so the sound is on long enough to hear.
1410 If we are chiming the hour, turn the sound off if it was on, and on if it was off. If we turned it off, that's one less chime to do.
1420 Likewise for the quarter-hour chime.
1800 Kill time till a new second comes around.
1810 Go back and redraw the hands appropriately.
===== End of main loop.

3000-3060
The code for the machine-language timer routine. The routine is called once every tick of the Atari's clock, which ticks every sixtieth of a second. The equivalent Basic code for it is:

```
            TICK=TICK+1 : IF TICK<60 THEN RETURN
TICK=0 : SEC =SEC +1 : IF SEC <60 THEN RETURN
SEC =0 : MIN =MIN +1 : IF MIN >59 THEN MIN=0
FIFTHHR=FIFTHHR+1 : IF FIFTHHR<12 THEN RETURN
FIFTHHR=0 : HR =HR+1 : IF HR  >59 THEN HR=0
RETURN
```

Note that HR holds where the hour hand is, not the actual hour of the day.

===== The subroutine at line 4000 stows the machine language away in a safe place.
4000 Make room to hold the hexadecimal ('hex') notation for the machine code.
4005 Say what's happening.
4010 Arrange to read the data at line 3000. J tells where the machine language goes. (For those of you who know something of the innards of the Atari, the code goes into the area where the IOCB's are kept, and so is safe. If you don't know what IOCB's are, this aside is just to calm the fears of those who know enough to worry about esoteric details.)
4020 Read the hex for one byte of code. If the byte is "==", we've run out of code, so stop already!
4030-4040
Convert the hex to decimal.

4050 Stuff the byte away, make J point to the next location for the code, and go back for the next byte.

4100 POKE #1: Turn off interrupts. POKE #2 and #3: Tell the Atari where the timer routine is. POKE #4: Turn interrupts back on. (For the novices, the interrupts are what run the Atari's clock, and care must be taken whenever they are meddled with, which is why they are turned off and on again. When they are off, we can meddle without fear. Even so, we must still be sure that our meddling really does what we wanted it to).)

4200 Go back to wherever we came from.

Novice Notes

In each issue of Iridis I'll cover one or more computer topics to help you understand what is going on inside your Atari.

As you probably know, computers are very good at doing what you tell them to do. A computer "program" is simply a detailed set of directions (or "recipe") you tell the computer to follow. Some folks who don't understand computers are in awe of how "smart" computers are. But computers aren't smart at all: they just follow directions! (I'll be the first to admit, however, that complex systems that include computer hardware and software can be impressive indeed. There is a significant debate among computer science types about what it takes before you can say that a computer system exhibits "intelligence", but that's another story.)

The programs in Iridis are mainly written in the computer language called "Basic". Basic is a fairly simple, easy-to-learn way to tell computers what it is you want them to do. There are many different versions or dialects of Basic. The most common versions on microcomputers (including the Commodore Pet, the Radio Shack TRS-80, and the Apple II) were all written by a company called Microsoft. There is a major difference between Atari's version of Basic and Microsoft's version: Atari's Basic handles strings much more awkwardly than Microsoft's. What is a 'string', you ask? In computerese, a string is some collection of letters, numbers, and punctuation, all wrapped up in a neat little package. For example, "How are you?" is a string. To store this string in our Atari, we'd do something like the following:

```
100 DIM EXAMPLE$(20)
110 READ EXAMPLE$
120 PRINT EXAMPLE$
500 DATA How are you?
```

In line 100 we tell the computer to reserve 20 locations in its memory for a character variable called EXAMPLE$. (Note that a trailing dollar-sign always means it is a character rather than a numeric variable.) In line 110 we tell the computer to go to the DATA statement and grab everything it finds (up to the next comma, or the end of the data) and put it in the variable EXAMPLE$. Finally, we have the computer print the contents of EXAMPLE$ on the screen in line 120. (Notice that there are no double quotes in the DATA statement. The double quotes are used only when the computer might mistake the string for something else. For example, assume the program contains a statement that says

```
130 EXAMPLE$="xxxxxxxxxxxxxxx"
```

If the double quotes weren't there, the Atari would try to follow the 'directions' after the equals sign. Since there are no directions there, things would fare rather badly.) In Microsoft Basic, you don't have to reserve space for character variables, as we did in line 100. You just go merrily on your way, and the system keeps track of things and grabs any space it needs along the way

Here are some examples of things you need to be able to do with characters, and how they are done on the Atari. Since lots of the programs published in computer magazines use Microsoft Basic, we've also shown how things are done that way.

In these examples, we assume that we've told the Atari how much space to set aside for OLD$ and NEW$ as follows:

```
100 DIM OLD$(50), NEW$(50), STUFF(50)
```

(In the examples, we never use 50 locations. It's just easy to go ahead and allow for a little more than we need.)

==========

To find out how long a string is:

```
OLD$ = "Welcome to Atari Basic."
HOWLONG = LEN(OLD$)
(HOWLONG will be 23.)
[This one is exactly the same for Microsoft!]
```

==========

To take the first character of a string:

```
OLD$ = "YES"
NEW$ = OLD$(1,1)
(NEW$ will now contain the letter 'Y'.)

[In Microsoft: NEW$ = LEFT$(OLD$,1) ]
```

To "glue" (concatenate) the characters in STUFF$ to those in OLD$ and put the result in NEW$:

```
OLD$   = "ABC"
STUFF$ = "DEFGH"
NEW$   = OLD$: NEW$(LEN(NEW$)+1) = STUFF$
(the result in NEW$ is "ABCDEFGH").

[In Microsoft: NEW$ = OLD$ + STUFF$ ]
```

==========

Now, let's grab some characters from the middle of a string:

```
OLD$ = "How are you?"
NEW$ = OLD$(5,9)
(the result in NEW$ is "are y").

[In Microsoft: NEW$ = MID$(OLD$,5,5) ]
```

==========

How about getting characters from the rightmost part of a string?

```
OLD$ = "Not too bad, thanks."
NEW$ = OLD$(14)
(the result is "thanks" in NEW$.)

 <In Microsoft: NEW$ = RIGHT$(OLD$,6)
                or: NEW$ = MID$(OLD$,14) >
```

==========

Above, we mentioned how to find the length of a string, without explaining what the length is. Essentially, the length of a string is the number of letters, digits, etc. that are in the string. Thus, the string "QUASIMODO" has a length of 9, since there are nine letters in it. In Atari Basic, you must be very careful not to confuse the length of a string with the size of the space given over to hold the string. The size of the space controls the longest string that can be held there, but the string may be any length up to that, even zero (a string with no letters in it at all). (A string with nothing in it is called a 'null string'. It is useful because there are times when you must, for example, print a string, but don't want to print anything. Printing a null string solves the problem. Since there's nothing in the string, nothing will print.)

drawn, it draws a horizontal line from the just-plotted point to the first colored spot it finds to the right. The result is that you get a pile of horizontal lines of the same color, filling in an area with one color.

Although Fill behaves much like DRAWTO, using it requires a different set of commands. To draw a line of color 1, you would say

```
COLOR 1 : PLOT OLDX,OLDY : DRAWTO NEWX,NEWY
```

(If the graphics cursor is already at OLDX, OLDY [if that's where the last point or line went to], you don't need the PLOT.) To do a Fill with color 3

```
COLOR 3 : PLOT OLDX,OLDY : POSITION NEWX,NEWY
POKE 765, 3 : XIO 18,#6,0,0,"S:" : PLOT NEWX,NEWY
```

The COLOR statement controls the color of the drawn line. (It doesn't have to be the same color as the filled area.) The PLOT tells where one end of the line is, and the POSITION tells where the other end is. (The line hasn't been drawn yet, though.) The POKE sets the color of the filled area. It behaves exactly like a COLOR statement, and takes the same set of numbers for colors. Then, the XIO draws the line and does the fill. Finally, a PLOT to the end of the line is needed to avoid a bug in Basic which leaves the graphics cursor somewhere other than the line end after a Fill. (The graphics cursor is just like a normal cursor, except that it's invisible. It tells where the last thing put on the screen went.)

===== Variables =====

C$	Holds the 'command' to the drawing subroutine
H	Horizontal destination for a line
V	Vertical destination for a line
HORG	Left side of drawing frame
VORG	Top side of drawing frame
C	Color of drawn lines
WASH	Color of filled areas (it's a painting term)

===== The Program =====

100	Make room for the Drawing Subroutine command
110	Set the Read Pointer to line 7000 (where the logo data is)
200	Go into high resolution graphics mode.
210-235	Choose the initial colors for the logo.
240	Make the cursor invisible, so we don't sully the text window.
250	Set the left margin (to make the text window pretty).

300 Go off to the Drawing Subroutine, and draw the logo.
350 Kill some time, to allow him to admire the outline of the logo.
400 (Go off to the routine that puts in colors in some arbitrary order.)
410 (Go off to the routine that puts in colors in the built-in order.)
500 Choose a random color, with a random brightness, and make the logo letters that color.
510 Kill some time to let him admire the pretty colors.
520 Go back and pick another color.
600 The color sequence is controlled by the data at line 700.
610 Read a color and brightness. If the color number is less than zero, we've run out of color sequence, so start over from the beginning.
620 Make the letters the appropriate color.
630 Kill some time.
640 Go back for the next color in the sequence.
790 Display all the colors in some semblance of order.
800 For each brightness possible,
810 and For each color possible,
820 Change the color of the letters
830 Kill some time
840 Do next color and brightness.
850 We ran out of colors, so start over.

===== The Drawing Subroutine =====

5000 Get the next drawing command. If the command is a number, it means 'draw a line to there', so go draw the line.
5010 Is the command 'P' (for Point or Plot)? If so, get the position of the dot, and put a dot there.
5015 Is the command 'O' (for Origin)? If so, get the new upper-left frame corner position.
5020 Is the command 'S' (for Stop)? If so, go away. Our job is done.
5030 Is the command 'C' (for Color)? If so, change both the line color and the wash color. (That way, filled-in areas will match the lines surrounding them automatically. If different colors are wanted (as with the logo), just use the Wash color command.)
5035 Is the command 'W' (for Wash color)? If so, get the new Wash color.
5040 Last chance: if the command isn't 'F' (for draw line with Fill), throw it out and try again.
5050-5060 Get the destination of the line, and do the fill: Position the graphics cursor at the other end of the line, tell the Atari the color to fill with (the wash), and draw the line (with filling).

5065 Patch over a bug in the Atari Fill command: the Atari, after a fill the graphics cursor is left at the end of the filling line (at the right end) rather than the end of the drawn line (at the left end). With this plot, we push the graphics cursor back to the end of the drawn line.

5070 Go back around for another command.

5100 We must draw a line. Get the numeric value of the 'command' (which was actually the first half of the position of the other end of the line), and then read the other half. Having the position, we need only draw a line.

5900 Take a position relative to the corner of the frame, and change it to a position on the screen.

Hacker's Delight

Hacker's Delight is a column dedicated to the confirmed hacker who wants to know everything possible about how his computer works. (Hacker, n., a person who uses computers as a hobby. Hackers are rarely content with the obvious ways to accomplish something, prefering instead to use strange and devious methods to a given end. A favorite pastime of hackers is to find out how to do things which are ostensibly undoable on their favorite computer. For example, how to get six or eight different colors on the screen at once, even though all the manuals claim that five is an absolute limit.) In this space, we intend to publish whatever information we can discover about the innards of the Atari, and how to make use of things undocumented. Everyone is invited to send us whatever they think might be of interest (and even that which might not - you never know when you'll want some odd piece of information). Occasionally, I'll toss out suggestions for things that ought to be discovered that, to our knowledge, haven't been (for example, how can you turn off the BREAK key, or even better, make it a TRAPable error?).

As a hacker, you have undoubtedly looked at Atari's cartridges and wondered how they can do all that, when you can't get any more than a GRAPHICS 7 area, without text in it, either. Your text is confined to a dinky four-line area at the bottom.

The truth is that, buried in the plastic case somewhere, there is a special IC, whose sole purpose is to maintain the screen display, with powers fully capable of Great Feats of Video Wizardry.

The full story of the Video Chip, as we shall call it, takes up far more space than we can give (and we don't know all there is to know about it yet). However, to start you off, here's how to get more graphics modes on the screen at one time (including a couple that Basic doesn't know about).

The Atari display is controlled by what may be called (borrowing a term from the computer graphics field) a 'display list'. That is, there is a chunk of memory set aside that tells exactly how the screen should look, where the stuff on the screen should come from, and so on. The heart of creating your own screen formats is knowing how to construct your own display list.

(Some definitions: a scanline is the thinnest possible line on the screen. It is one dot high. Any other line is a line composed entirely of one graphics mode. Also, the bits in a byte are numbered from 0 to 7, starting from the right, as 7 6 5 4 3 2 1 0. In other words, a byte with only bit 7 on equals 128, or hex 80.) The general format of a display list is: a command that tells where to get the screen contents from, followed by a series of bytes telling how to display the contents, then another memory address, then more formatting bytes, another address, more formatting bytes, and so on, to the end of the list. Naturally, there need only be one address, if you want to display only one area of memory (as in text mode). The formatting bytes do NOT all have to specify the same graphics mode. Every one of them can, if you like, specify a different mode. The chief restriction is that the graphics mode selected by one formatting byte extends to both sides of the screen, so you can cut the screen only into horizontal pieces, not vertical ones.

In the display list, the bits in a byte have the following meanings:

Bit 7 - Bit 7 has no visible effect.
Bit 6 - the Address bit - If this bit is on, the next two bytes contain the address from which the screen contents should be taken. If this bit is off, the memory displayed comes immediately after the memory displayed by the previous byte.
Bit 5 - Not known - the apparent effect is this: The line for a byte with bit 5 off gets mashed flat, but only if the byte before it had bit 5 on. The line is still there, but it takes only one scanline. Most peculiar. (Someone tell us what this bit really does, please!)
Bit 4 - Another unknown bit. The apparent effect of this one is to make the line occupy 20% more bytes than it normally would, but to ignore the first and last 10%. That is, the visible part is still x bytes wide, but there are 0.1x invisible bytes on either side of the visible part. (Another mystery begging to be solved.)

Bits 3-0 -
the graphics type of this line of the screen. (I don't call it the graphics mode to avoid confusion.)

There are fourteen graphics types available:

2 Normal text-mode display (one line's worth)
3 Text mode, but with an extra two scanlines after the text. Also, some characters have the dots scrambled some.
4 A most peculiar mode. It is a text mode of sorts, but the letters have colored fringes around them.
5 Same as type 4, but with the letters twice as high
6 Graphics mode 1 (again, for one line of text).
7 Graphics mode 2
8 Graphics mode 3
9 Graphics mode 4
10 Graphics mode 5
11 Graphics mode 6
12 Graphics mode 6, but with the dots only half their usual height.
13 Graphics mode 7
14 Graphics mode 7, half-size dots
15 Graphics mode 8

Normally, graphics types 8 and 9 use (and display) 10 bytes from the screen memory. Types 6, 7, 10, 11, and 12 use 20 bytes, and the rest use 40. If bit 4 is on, types 8 and 9 use 12 bytes, of which the middle 10 are displayed. Types 6, 7, 10, 11, and 12 use 24 bytes, displaying the middle 20. The rest use 48 bytes, with the middle 40 displayed.

Types 0 and 1 don't appear in the list because they are not really graphics types. They cause the Atari to do things other than display memory. Moreover, in them bits 6, 5, and 4 lose their usual meaning. Type 0 produces lines of the background color (register 4). Bits 6, 5, and 4, treated as a single, three-bit number, tell how many scanlines high the line is. The height is the number plus 1, so that 000 is one scanline, 011 is four scanlines, and so on. No memory is used by a type-0 byte.

Type 1 seems to be the special-function type. Type 1 with bit 6 on (hex 41) marks the end of the display list. Type 1 with bit 6 off means 'display list continued over there'. It is followed by the (two-byte) address of where the display list continues. The meanings (if any) of bits 4 and 5 for type 1 are unknown.

The normal text-mode display list (as you no doubt assume from the above, there is a different display list made for each graphics mode) looks like this (in hexadecimal):

```
70 70 70 42 40 5C 02 02 02 02 02 02 02 02 02 02
02 02 02 02 02 02 02 02 02 02 02 02 02 41
```

The first three bytes are type-0 (background) bytes, each specifying eight scanlines of background, for a total of twenty-four. This is the top border of the text area. Next is a type-2 (text) byte, but with the Address bit (bit 6) on, so the two bytes after it (40 5C) say where the screen memory is. This displays the first line of text in the text area. Then there are twenty-three more type-2 bytes, for the other twenty-three lines of text. Finally, there is a type-1 (special) byte, with bit 6 on, so that is the end of the display list.

Another display list: this one is for graphics mode 2, with a text window.

```
70 70 70 47 70 5E 07 07 07 07 07 07 07 07 07 42
60 5F 02 02 02 41
```

First there are the same three type-0 bytes, for the top border. Then we have a type-7 (GR.2) byte, with the address bit on. The next two bytes are the address for the screen memory. Next are nine type-7 bytes, the rest of the graphics area. Next is a type-2 byte, with address bit again (and two address bytes). That starts the text area. Three more type-2 bytes fill out the text area, and again, a type-1 byte to end the display list.

The GRAPHICS 3 display list, which ZAP uses, looks just like the graphics mode 2 list, except that type-8 bytes are used, and there are more of them. What ZAP does to get big letters is this: it searches the display list to find the type-2 byte marking the start of the text window. That byte, and the other three type-2 bytes following it, are changed into type-6 bytes, forcing the same memory to be displayed as large letters.

The other bit of information needed is how to FIND the display list. 'Tis simple: locations 560 and 561 contain the address of the display list. In other words, DISPLAYLIST = PEEK(560) + 256*PEEK(561). Then, to print out the display list:

```
100  I = 0
110  PRINT PEEK(DISPLAYLIST+I)," ",
120  IF PEEK(DISPLAYLIST+I)<>65 THEN I=I+1:GOTO 110
```

Feel free to try anything you like with the display list. Since it's all software or electronics, you can't hurt anything with a nonsensical display list (I speak from experience). Also, if you get yourself into a corner, [RESET] will get you out, restoring everything to normal.

Zap

ZAP is an addictive game in which you control a worm which, naturally enough, is going about in search of food. The program begins in an 'attract mode', playing the game by itself. To play it yourself, plug a joystick into jack 1 (the leftmost one), press the [START] button, and go to it. (Many people find that the joystick can be more reliably controlled if it is on a hard, flat surface, like a table or a hardcover book.)

The rules are these: You control the worm, maneuvering it about the screen with the joystick. The blue dots are food, and you get points for hitting (and eating) them. (There are other creatures about, none of which are shown, which are also eating the blue things.) The red lines are walls, which you bounce off of. The only fatal act is to run into yourself. The worm, being as bright as most worms are, starts chomping on itself, committing suicide. (If you run into a wall head-on, you will bounce off, but right into yourself.)

For each food particle the worm eats, you get 10 points, plus a bonus of one percent of your current score. Also, the worm's body grows three new segments. Each time something else eats a food particle, you lose 2 points. You are allowed to crash (into yourself) five times. After the fifth crash, the game is over.

Between games, ZAP returns to attract mode, displaying both the most recent score (labelled LAST), and the best score since the program was started (labelled BEST). To start a new game, press [START] again.

```
0 REM ZAP
1 REM COPYRIGHT(C) 1980 IRIDIS
2 REM BOX 550, GOLETA, CA. 93017
3 REM ALL RIGHTS RESERVED
10 REM AS OF 5 MAR 80
100 GOTO 10000
1000 REM * UPDATE TARGETS *
1010 REM ERASE A TARGET
1020 IF DOTS<MINDOTS OR RND(1)>DOTCHANCE THEN 1100
1030 DOTS=DOTS-1:LASTDOT=LASTDOT+1:IF LASTDOT>MAXDOTS THEN LASTDOT=1
1040 R=DOTROW(LASTDOT):C=DOTCOL(LASTDOT):LOCATE C,R,I:REM CHECK WHERE DOT WAS
1050 REM IF STILL DOT, ERASE IT, AND LOWER SCORE
1060 IF I=DOT THEN COLOR BLACK:PLOT C,R:IF PTS>0 THEN PTS=PTS-2
1100 REM PLACE A TARGET
1110 IF DOTS>=MAXDOTS OR RND(1)>DOTCHANCE THEN 1200
1120 REM FIND EMPTY SPOT
1130 R=INT(RND(1)*(MAXROW-MINROW-1))+MINROW+1
1135 C=INT(RND(1)*(MAXCOL-MINCOL-1))+MINCOL+1
1140 LOCATE C,R,I:IF I<>BLACK THEN 1120
1150 DOTS=DOTS+1:FIRSTDOT=FIRSTDOT+1:IF FIRSTDOT>MAXDOTS THEN FIRSTDOT=1
1160 DOTROW(FIRSTDOT)=R:DOTCOL(FIRSTDOT)=C:COLOR DOT:PLOT C,R
1200 RETURN
2000 REM +----------------------+
2010 REM |THINGS WHICH HAPPEN    |
2020 REM |ONCE PER WORM MOVEMENT|
2030 REM +----------------------+
```

```
2040 IF NOISE0>0 THEN NOISE0=NOISE0-1:IF NOISE0=0 THEN SOUND 0,0,0,0
2050 IF NOISE1>0 THEN NOISE1=NOISE1-1:IF NOISE1=0 THEN SOUND 1,0,0,0
2100 REM SHORTEN TAIL OF WORM
2110 IF SEGS<MINSEGS THEN 2200
2120 SEGS=SEGS-1:LASTSEG=LASTSEG+1:IF LASTSEG>MAXSEGS THEN LASTSEG=1
2130 COLOR BLACK:PLOT SEGCOL(LASTSEG),SEGROW(LASTSEG)
2200 REM LENGTHEN HEAD OF WORM
2210 IF SEGS>MINSEGS THEN 0
2220 REM SEE WHERE HE WANTS TO GO
2230 REM IGNORE HIM IN ATTRACT MODE
2240 I=STICK(0):IF ATTRACT THEN I=15
2250 IF I<>15 THEN HSPEED=HCHANGE(I):VSPEED=VCHANGE(I):POKE 77,0
2260 REM DETERMINE NEW HEAD POSITION
2270 R=ROW+VSPEED:C=COL+HSPEED:LOCATE C,R,I
2280 REM CAN HE GO THERE?
2290 IF I<>EDGE THEN 2350
2300 REM HE HIT THE BORDER. BOUNCE.
2310 IF R<=MINROW OR R>=MAXROW THEN VSPEED=-VSPEED
2320 IF C<=MINCOL OR C>=MAXCOL THEN HSPEED=-HSPEED
2330 SOUND 1,30,10,6:NOISE1=1
2340 GOTO 2260
2350 REM HE CAN GO THERE.
2360 ROW=R:COL=C:COLOR SEGMENT:PLOT COL,ROW
2380 FIRSTSEG=FIRSTSEG+1:IF FIRSTSEG>MAXSEGS THEN FIRSTSEG=1
2390 SEGROW(FIRSTSEG)=ROW:SEGCOL(FIRSTSEG)=COL:SEGS=SEGS+1
2400 REM BUT IS IT WISE?
2410 IF I<>DOT THEN 2450
2420 REM HE HIT A TARGET!
2430 SOUND 0,0,0,0:PTS=PTS+10+INT(0.01*PTS):MINSEGS=MINSEGS+3
2440 SOUND 0,33,12,6:NOISE0=2:GOTO 2500
2450 IF I<>SEGMENT THEN 2500
2460 REM HE HIT HIMSELF!
2470 SOUND 1,0,0,0:REM DO CRASH SOUND
2480 FOR I=0 TO 3:SOUND 0,29,0,15-4*I:FOR J=0 TO 10*2^I:NEXT J:NEXT I
2490 SOUND 0,0,0,0:CRASH=1:GOTO 2510
2500 GOSUB 1000
2510 RETURN
10000 REM +----------------------+
10010 REM |ONE TIME INITIALIZATION|
10020 REM +----------------------+
10030 REM * LIMITS AND SIZES *
10040 MAXDOTS=20:REM # TARGETS ALLOWED
10045 MINDOTS=5:REM DON'T DELETE DOTS UNLESS MINDOTS ON SCREEN
10050 MAXSEGS=250:REM LONGEST POSSIBLE WORM
10060 MAXCRASHES=5:REM CRASHES PER GAME
10070 REM * SCREEN CONTROL *
10080 GRAPHICS 3:SETCOLOR 2,0,0
10090 MINROW=0:MINCOL=0:MAXROW=19:MAXCOL=39
10100 REM DIDDLE VIDEO DISPLAY LIST
10110 REM TO PRODUCE BIG LETTERS
10120 SCRNMAP=PEEK(560)+256*PEEK(561)
10130 IF PEEK(SCRNMAP)<>66 THEN SCRNMAP=SCRNMAP+1:GOTO 10130:REM FIND TEXT
10140 REM MAKE LETTERS BIG
10150 POKE SCRNMAP,70:POKE SCRNMAP+3,6:POKE SCRNMAP+4,6:POKE SCRNMAP+5,6
10160 REM * COLORS (AND REGISTERS) *
10170 BLACK=0:SEGMENT=1:DOT=2:EDGE=3
10180 SETCOLOR 4,0,0:SETCOLOR 0,0,10
10190 SETCOLOR 1,8,4:SETCOLOR 2,2,2
10200 REM * DIMENSIONING ARRAYS *
10210 DIM DOTROW(MAXDOTS),DOTCOL(MAXDOTS):REM KEEPS TRACK OF TARGETS
10220 DIM SEGROW(MAXSEGS),SEGCOL(MAXSEGS):REM KEEPS TRACK OF WORM
10230 DIM BL$(10):REM FOR ALIGNING NUMBERS
10240 DIM HCHANGE(15),VCHANGE(15):REM STICK->DIRECTION TABLES
10300 REM * ODDMENTS *
10310 POKE 82,0:REM LEFT MARGIN TO ZERO
10320 POKE 752,1:REM TURN OFF CURSOR
10330 BESTPTS=-1:REM BEST SCORE SO FAR
```

```
10340 ATTRACT=1:REM START IN ATTRACT MODE
10350 BL$="          {ESC}":REM LEADING BLANKS FOR NUMBERS
10400 REM SET UP JOYSTICK TO DIRECTION TABLES
10410 FOR I=0 TO 15:READ C,R:HCHANGE(I)=C:VCHANGE(I)=R:NEXT I
10420 DATA 0,0, 0,0, 0,0, 0,0, 0,0, 1,1, 1,-1, 1,0
10430 DATA 0,0, -1,1, -1,-1, -1,0, 0,0, 0,1, 0,-1, 0,0
11000 REM +----------------------+
11010 REM |PER-GAME INITIALIZATION|
11020 REM +----------------------+
11030 PTS=0:REM SCORE SO FAR IN GAME
11040 CRASHES=0:REM NO CRASHES YET
11500 REM +------------------------+
11510 REM |PER-CRASH INITIALIZATION|
11520 REM +------------------------+
11525 FOR I=1 TO 200:NEXT I
11530 FIRSTDOT=0:LASTDOT=0:DOTS=0:REM NO TARGETS
11540 FIRSTSEG=0:LASTSEG=0:SEGS=0:REM NO WORM
11550 MINSEGS=16:REM INIT LENGTH 16
11560 ROW=MAXROW-1:COL=MAXCOL-1:HSPEED=-1:VSPEED=-1:REM INIT DIR AND SPEED
11600 REM DRAW BOARD
11610 PUT #6,125:COLOR EDGE
11620 PLOT MINCOL,MINROW:DRAWTO MINCOL,MAXROW:DRAWTO MAXCOL,MAXROW
11630 DRAWTO MAXCOL,MINROW:DRAWTO MINROW,MINROW
11650 REM * SET UP INITIAL TARGETS *
11660 DOTCHANCE=1
11670 IF DOTS<MINDOTS THEN GOSUB 1000:GOTO 11670
11675 FOR I=1 TO 100:NEXT I
11680 DOTCHANCE=0.1
11690 CRASH=0:REM NOT YET CRASHED
11700 IF ATTRACT=0 THEN 11900
11710 IF BESTPTS>=0 THEN 11750
11720 PRINT "{CLEAR} ZAP (C)1980 IRIDIS"
11730 PRINT "PRESS start TO PLAY. (NEEDS A JOYSTICK)"
11740 GOTO 11800
11750 PRINT "{CLEAR}       last       best";
11760 PRINT BL$(LEN(STR$(LASTPTS)));LASTPTS;
11770 PRINT BL$(LEN(STR$(BESTPTS)));BESTPTS;
11780 PRINT "_____PRESS start TO BEGIN"
11800 IF PEEK(53279)<>7 THEN ATTRACT=0:GOTO 11000
11810 GOSUB 2000:IF CRASH THEN 11500
11820 GOTO 11800
11900 PRINT "{CLEAR 2 DOWN}";
12000 PRINT "{2 UP}SCORE: ";PTS;" "
12010 PRINT "CRASHES: ";CRASHES
12020 GOSUB 2000:IF CRASH=0 THEN 12000
12100 CRASHES=CRASHES+1:IF CRASHES<5 THEN 11500
12110 LASTPTS=PTS:IF PTS>BESTPTS THEN BESTPTS=PTS:REM GAME OVER
12120 ATTRACT=1:GOTO 11000:REM BACK TO ATTRACT MODE
```

Behind the Scenes

The ZAP program keeps track of the positions of the worm segments and the targets in a 'queue'. A queue is something that you put things into it at one end, and take them out at the other, like a line at a grocery store (the British, in fact, refer to those lines as 'queues' [the computer people stole the word from them]).

Since queues contain a bunch of things, the obvious way to keep them is in an array, since arrays exist to hold bunches of things. (Horizon-broadening department: Arrays are not the only way to keep queues. Some languages other than Basic allow you to do queues in other ways. All of the other ways can be force-fit into Basic, but it isn't really worth it. [A true-life example of how some languages are better at various tasks than others. Basic isn't very good at queue-handling.]) When you store a queue in an array, there are two ways to put things in and take them out. You can keep the ends of the queue in a fixed place, and move the items around in the array (e.g. after taking item 1 out of the queue, you move item 2 to where item 1 was, item 3 to where item 2 was, and so forth). Alternatively, you can keep the items where they're put, and move the ends around (when you take item 1 out, you leave items 2 on alone, but remember to ignore item 1 in the future). In either method, you need to keep track of where in the array to put the next thing that is put into the queue (you need to know where the line ends).

At first glance, the first method is better. Suppose your array can hold twenty items. If you use the second method to keep the queue, it would seem you'd run out of room in the array. Even if you ignore the removed items, they still take up space. Remove twenty items, and the array is full of ignored items. If you always move the items into the vacancies, you won't run out of room until the array is full of wanted items. As it turns out, if you use a trick, the second method is easier. The trick allows you to re-use the ignored items. What you do is to pretend that the ends of the array are connected, so your array is a circle, like a roulette wheel. When either end of the queue falls off one end of the array, make the other end of the array the end of the queue. The Basic code to put something into the queue is simple:

```
BACK=BACK+1: IF BACK>ARRAYSIZE THEN BACK=0
ARRAY(BACK) = whatever
```

(BACK gets set to zero because arrays start at zero. Why waste a perfectly good array item?) Taking something off the queue is just as simple.

```
FRONT=FRONT+1 : IF FRONT>ARRAYSIZE THEN FRONT=0
whatever = ARRAY(FRONT)
```

Care must be taken that BACK and FRONT never pass each other, but other than that everything works nearly automatically. Note that if BACK and FRONT are equal, the queue is empty.

	===== Arrays =====
DOTROW()	and
DOTCOL()	the target queue
SEGROW()	and
SEGCOL()	the worm segment queue. Remember that the worm is stored backwards, with the head of the worm at the back of the queue, and the tail of the worm at the front of the queue. This is because we want to erase the worm tail-first.
BL$()	holds blanks for lining up numbers
HCHANGE()	and
VCHANGE()	tell how much a position changes when it moves in one of eight directions. (They are dimensioned to 15 because the joystick can return numbers up to 15.)
	===== Variables =====
ATTRACT	says whether we're in attract mode or not
BESTPTS	the best score since the program was started
CRASH	notes if there's been a crash lately
CRASHES	how many crashes so far this game
DOTCHANCE	how likely a new target is
LASTPTS	score for just-ended game
MINSEGS	how long the worm is right now
PTS	the current score in the game
SCRNMAP	tells where the screen display list is kept
ROW	and
COL	where the head of the worm is
HSPEED	and
VSPEED	what direction the worm is going
FIRSTSEG	where the head of the worm (the back of the worm segment queue) is.
LASTSEG	where the tail of the worm (the front of the worm segment queue) is
SEGS	how many segments there are in the worm at present
FIRSTDOT	where the newest target (the back of the target queue) is
LASTDOT	where the oldest target (the front of the target queue) is
DOTS	how many targets are on the screen (counting those which have been hit by the worm, since they're still in the target queue).
	===== Constants =====
MAXDOTS	maximum number of targets on the screen at one time
MINDOTS	minimum number of targets on-screen (there may actually be fewer, but ZAP won't remove any unless there are at least that many)
MAXSEGS	maximum number of segments to a worm
MAXCRASHES	number of crashes allowed per game

MINROW and
MAXROW tell the top and bottom rows on the board
MINCOL and
MAXCOL tell the leftmost and rightmost columns

BLACK the color number for the background
DOT the color number for the targets
SEGMENT the color number for the worm
EDGE the color number for the border of the board

===== The program =====
(REM statements will be ignored whenever it's expedient.)

100 Skip around the subroutines. They are first to cut down on the time needed to find them on GOSUBs.

1000 == Target handler ==

1020 If there are too few targets on the screen already, or if we don't feel like it, skip erasing a target, and go see about putting another on the screen.

1030 Note that there will be one less target on the screen. Take the oldest target off the queue.

1040 See what is where we put the target.

1060 If there's still a target there, he missed it. Take it off, and penalize him for missing it.

1100 How about putting up a target?

1110 If there are already enough targets, or we don't feel like it, don't put a target up.

1130 Pick a location for the target.

1140 Take a look at the proposed location to see if it's empty. If it isn't, go back and pick a different location.

1150 Note that there will be one more target on the screen. Make room in the queue for the new target.

1160 Put the target in the queue, and on the screen.

1200 Go back to whatever you were doing before you starting fiddling with targets.

===== The main subroutine, which moves the worm, fiddles with the targets, and does most other things that need doing.

2040 If there's a sound on channel 0, and it's time to turn it off, do so.

2050 Likewise with channel 1.

2100 == Move the worm

2110 If the worm is shorter than it should be (he hit a target lately, perhaps), don't remove the tail end. Go directly to where it grows.

2120 Note that the worm is one segment shorter. Take the tail segment off the queue.

2130 Erase the last segment.

2210 If the worm is longer than it should be, leave it alone. (It should never happen, but one of the prime Rules of Programming is "Don't take chances". If you assume it won't happen, it will. (Umpteenth corollary to Murphy's Law.))

2240 See which way the joystick's been pushed. If we're in attract mode, we don't care about the joystick, so pretend it's straight up.

2250 If the joystick is pushed to one side, adjust the worm's path appropriately, and turn off glitch mode (see elsewhere for what glitch mode is).

2270 Compute the new position of the worm's head, and see what's there now.

2290 If it's not the edge of the board, go check for something else (at line 2350)

2300 It's the edge of the board. Make him bounce.

2310 If he hit a top or bottom wall, send him up if he's going down, and vice versa.

2320 If he hit a side wall, send him left if he's going right, and vice versa.

2330 Turn on the bounce sound. Note that we did so, so it can be turned off later.

2340 Now that the bounce has been done, go back and see where he bounced to.

2350 It's not the wall he hit.

2360 Remember his new location, and put new segment on the screen.

2380-2390
Record new segment in the queue, and note that there's one more segment to the worm.

2400 Now we see if his new position is hazardous.

2410 If he did NOT hit a target, go off to see what he did hit.

2430 Turn off the target sound, in case it's still on from a previous target. Increase his score, as well as his length.

2440 Turn on the target sound, and make a note that it's on. Go and fiddle with the other targets.

2450 It's not a target, either. Was it.... himself? If not, go play with the targets.

2460 He blew it! The klutz ran into himself.

2470 Turn off bounce sound.

2480 Make the crash sound (Esoterica department: it's white noise with an exponetial decay.)

2490 Note that he just crashed, and go back to whatever we were doing before we started all this.

2500 Go fiddle with the targets, perhaps removing one, perhaps putting one up.

2510 Go back to whatever we were doing before.

===== Set up things that stay set

10000-10060
Set limits on sizes of arrays, the worm, the number of targets, etc.

10080 Choose the graphics mode with the biggest squares, and make the text window black.

10090 Remember the size of the screen.

10120 Find the video display list.

10130 Search the display list for the start of the text window.

10150 Change the text window into a GRAPHICS 1 window.

10170-10190
Choose the colors for everything.

10200-10240
Create all the arrays and strings we need.

10310 Set the left margin to zero.

10320 Turn off the cursor.

10330 No best score as yet.

10340 Go into attract mode, to tempt him into playing.

10350 Set up some leading blanks, for right-justifying numbers in the text window. There is a trailing {ESC} in the string. This is because Atari Basic doesn't permit taking substrings with no characters in them. So, instead of taking substrings with zero to nine characters in them, we take substrings with one to ten, and have the last character be one that is invisible, so the effect is as if we had one less character.

10400-10430
Set up the tables that tell how much to change the position of the head of the worm for each way the joystick can be held. Most of those 0,0 entries are for codes that the STICK() function can't return.

11000 ===== Set things that need to be re-set before each game.

11030 No score, as yet.

11040 No crashes yet, either.

11500 ===== Set things that need to be re-set after each crash. (We pretend that there was a crash just before each game.)

11525 Kill a little time, just to let the player realize that he's crashed. (People just don't react as fast as computers do.)

11530 Dispose of any remaining targets.

11540 Dispose of any remaining worm, for that matter.

11550 Create a new worm, whose length is 16 segments.

11560 Start it off heading to the upper left, from the lower-right corner of the board.

11610 Erase the old board (and the text window with it, unfortunately). Set the next-drawn lines to be the color of the edge of the board.

11620-11630
Draw the edge of the board.

11650-11675
Call the target-fiddling routine until there are enough targets on the board to permit play. DOTCHANCE is set to 1 so we don't waste time letting the routine decide it doesn't want to put up targets.

11675 More time killing, to let the player realize there's a new board already.

11680 Let the target-fiddling routine skip putting up targets again. With DOTCHANCE set to 0.1 (1/10), it will put up a new target one time in every ten it is called, on the average.

11690 Say that he hasn't crashed yet.

11700-11710
Choose what to say in the text window. If we're in the midst of a game, give the score. If we're in attract mode, but after a game, give both the latest and best scores. Otherwise, we're just starting off, so give the name and copyright notice.

11720-11740
Tell people what we are, and who owns us.

11750 Print the last and best score labels.

11760 Print the latest score. We print an appropriate number of blanks to line the number up with its label. The substring starts with the length of the printed number, so we'll get fewer blanks as the number gets longer.

11770 Print the best score.

11780 Put up a divider, and tell the player how to start a new game.

11800 We're in attract mode. Peek at the [START] button, and if it is down, drop out of attract mode, and start a real game. (Actually, the peek checks only if any of [START], [OPTION], or [SELECT] is pressed. Atari Basic has no convenient way of picking [START] from the others, so we look for any of them. After all, [START] and [SELECT] might be pressed at the same time, and that should count as pressing [START].)

11810-11820
Move the worm one position. If it crashed (and it will, sooner or later), start the game over. Otherwise, move it again.

11900-12010
At last! A real game! Print the score, and how many crashes there were.

12020 Keep moving the worm until it crashes. (Again, eventually it will.)

12100 Tally up another crash. If he has more lives left, set him up another worm, and let him go again.

12110 Note score for the just-finished game, and see if we have a new record score.

12120 Since game is over, we go back into attract mode.

Oddments

ODDMENTS is the repository for all the facts, fancies, and rumors that don't warrant an article of their own. We encourage contributions, and will acknowledge anything we see fit to use. Here's your chance to see your name in print.

▲ After the advent of video games, it was discovered that, when they were left on too long, a picture of the game was 'burned' into the screen, so you saw it even when you weren't playing the game. The Atari has a feature which is intended to prevent this from happening with it. After about nine minutes go by without anything happening at the keyboard, the Atari starts (temporarily) changing the color registers (and therefore the colors on the screen.) This prevents any one phosphor (the stuff that glows) from getting over-used.

We have, for convenience, declared that the Atari is in 'glitch mode' when it is changing the colors for you. (The name arose because the color-changing wasn't documented when it first happened to us, and it appeared that the Atari had a glitch in it somewhere.) If you have a program which uses joysticks or paddles exclusively, the keyboard may well go unused for the requisite nine minutes, allowing the Atari to go into glitch mode. You can prevent that by POKEing a zero into location 77 every so often (less than nine minutes). (Location 77 is the timer for glitch mode. Putting a zero in it resets the timer back to the start.)

▲ When doing such manipulations of strings as are possible, it is sometimes convenient to be able to make a string a specific length. To make a string N characters long, assign some SINGLE character to the Nth character of the string. For example, to make X$ 11 characters long, type

```
X$(11) = "A"
```

If you don't want to change the string, assign the current Nth character:

```
X$(11) = X$(11,11)
```

▲ For those who haven't been raised with (by?) computers: the control key (marked CTRL on the Atari) is a special kind of shift key. When it is pressed, any other key that is pressed at the same time will produce a different character from normal. For example, when the Atari is in lower-case mode, pressing the 'A' key gives you a lower-case A. Shifted 'A' gives you an upper-case A. Control-A (or ctrl-A, as we write it everywhere else) gives you a sideways T (one of the Atari's funny characters). The control key pressed by itself does nothing, any more than the shift key, pressed by itself, does.

Polygons

POLYGONS is entertainment, nothing more and nothing less. It does show off the high resolution graphics capabilities of the Atari rather nicely, and is fun to watch. It will not balance your checkbook, compute the interest on your Swiss bank account, or make you a better person. We think you'll enjoy it anyway.

```
0 REM POLYGONS
1 REM COPYRIGHT(C) 1980 IRIDIS
2 REM BOX 550, GOLETA, CA. 93017
3 REM ALL RIGHTS RESERVED
10 REM AS OF MARCH 5 1980
90 GOSUB 30000
100 MAX=32:MIN=6:DELAY=50:LET CONTRAST=4:INTERVAL=1:START=1
110 DIM R(2*MAX),C(2*MAX):DEG
120 CIRCLE=360
130 GRAPHICS 8+16
140 SETCOLOR 2,0,0
150 SETCOLOR 4,0,0
160 SETCOLOR 1,0,14
170 COLOR 1
180 KEYCODE=764
185 SPKR=53279
300 N=INT(RND(1)*(MAX-MIN))+MIN
310 FOR I=0 TO N-1
320 R(I)=95*SIN(CIRCLE*I/N)+96
325 C(I)=95*COS(CIRCLE*I/N)+160
330 R(I+N)=R(I):C(I+N)=C(I)
335 NEXT I
340 FOR I=N TO MAX:REM KILL SOME TIME
345 T=SIN(I):T=COS(Y)
350 NEXT I
355 FOR I=1 TO DELAY:NEXT I
400 I=INT(RND(1)*16)
410 B1=INT(RND(1)*8)*2:B2=INT(RND(1)*8)*2
420 IF ABS(B1-B2)<CONTRAST THEN 410
450 GOSUB 900:IF T=255 THEN 470
460 I=I+1:IF I>100 THEN POKE SPKR,0:I=0
465 GOSUB 900:IF T=255 THEN 460
470 PUT #6,125
475 SETCOLOR 4,I,B1
480 SETCOLOR 2,I,B1
485 SETCOLOR 1,I,B2
600 FOR SKIP=START TO INT(N/2) STEP INTERVAL
610 IF RND(1)>0.5 THEN 650
620 FOR I=0 TO N-1
630 PLOT C(I),R(I):DRAWTO C(I+SKIP),R(I+SKIP)
640 NEXT I
650 NEXT SKIP
660 POKE 77,0:REM KILL GLITCH MODE
670 FOR I=1 TO 50:POKE SPKR,0:NEXT I
680 GOTO 300
900 T=PEEK(764):POKE 764,255:RETURN
30000 DIM CR$(1):CR$=CHR$(155)
30010 GRAPHICS 2:OPEN #1,4,0,"K:":POKE 752,1
30020 SETCOLOR 0,8,12:SETCOLOR 3,9,4:SETCOLOR 2,0,0
30030 PRINT #6;CR$;CR$;CR$;CR$;
30040 PRINT #6;"    {12 C}"
30050 PRINT #6;"    {C} POLYGONS {C}"
30060 PRINT #6;"    {12 C}"
30070 PRINT "{DOWN}     COPYRIGHT (C) 1980  IRIDIS"
30080 PRINT "{DOWN}       PRESS RETURN TO BEGIN.";
30090 GET #1,T:CLOSE #1:CLR :POKE 752,0:GOTO 100
```

Behind the Scenes

Basically, POLYGONS figures out where the corners of the polygon are, and then draws lines between them. The lines are drawn in sets, each set connecting all the corners that are x corners apart (where x may be any number up to the total number of corners). When that set is done, POLYGONS proceeds to draw lines between corners that are x+1 corners apart, and so on.

===== Arrays =====

R()	holds the row (vertical) positions of the polygon corners
C()	holds the column (horizontal) positions of the polygon corners

===== Variables =====

MIN	least number of sides on a polygon
MAX	greatest number of sides on a polygon
DELAY	time to kill after calculating corners of polygon
CONTRAST	minimum contrast between lines and background
B1	brightness of background
B2	brightness of lines
N	number of points in polygon being drawn
INTERVAL	after drawing lines between corners that are x corners apart, draw lines between ones that are x+INTERVAL corners apart.
START	don't draw lines between corners that are less than START corners apart
SKIP	draw this set of lines between corners that are exactly SKIP corners apart (i.e. skip SKIP corners)

===== Constants =====

KEYCODE	where, in memory, the code of the last key press is kept
SPKR	where to poke to make the built-in speaker click
CIRCLE	the circumference of a circle, in radians

===== The Program =====

100	Set up various initial values for the variables.
110	Create the arrays R() and C(). They are 2*MAX items long because the corner positions are stored twice, to make the drawing easier.
120	Calculate the size of a circle.
130	Select the graphics mode we want (highest resolution + full screen)
140-160	Make the screen black. (Setting color register 1 doesn't really do anything here, but is included for completeness.)
170	Make drawn lines a different color from the background. (Which color depends on the color register settings.)
180	Note where to find key-pressed code.

190 Note where to stuff a zero to make the speaker click.
===== The main loop, choosing and drawing polygons
300 Pick how many sides (and corners) the polygon will have. A number between MIN and MAX will be chosen.
310-335 Figure out where the corners will be, and remember the positions. The +96 and +160 are to center the polygon on the screen.
340-350 Kill enough time so that every time we produce a new polygon, it takes about the same amount of time, however many sides it may have.
400 Pick a color for the background (and lines, since they're the same in graphics mode 8).
410 Pick a brightness for the background, and another for the lines.
420 If the two brightnesses are too close together, throw 'em out. Go back to line 410 to pick two more brightnesses.
450 Take a look and see if he pressed a key. (If T<>255, he did.) If he didn't, go off and draw the new polygon. If he did, leave the old polygon up until he presses another key.
460-465 Wait until he presses the other key. Make the speaker click every so often to remind him that we're waiting for him to do something.
470 Erase the screen.
475-485 Set the colors of the lines and background as chosen earlier.
===== The polygon-drawing loop starts here. Overall, the loop finds corners of the polygon SKIP points apart, and draws a line between them.
600 Select which set of lines we'll even consider drawing.
610 Don't draw some of the lines (for variety). Effectively, it's heads we draw, tails we don't.
620-640 Go to each corner of the polygon, and draw a line to the corner SKIP points away, clockwise.
650 Add INTERVAL to SKIP. If we haven't connected all the points, go back and connect the ones we missed.
660 Kill glitch mode. (See elsewhere for what glitch mode is.)
670 Make the speaker buzz, to say the polygon is done.
680 Go back and create another polygon.
===== Subroutine to check for a keypress
900 See if a key was pressed, and remember which one, if any. Then tell the Atari that, actually, nothing was pressed. (The Atari is gullible about that.)

About IRIDIS Listings . . .

As you might have noticed, the Atari has a number of peculiar characters that only the Atari can print. Needless to say, that presents us with a problem: we intend to print listings, and listings tend to be full of characters we can't print. As a way out, we have come up with a set of conventions for displaying those unprintable characters. (We must acknowledge our debt to the People's Computer Company, publishers of Recreational Computing, whose similar notation for the Commodore Pet's funny characters gave us the inspiration for our notation for the Atari's characters.)

The notation we will use has two basic rules: anything underlined is in reverse video, and anything in braces ({}) is special. Anything else you see is just what it appears to be.

The characters in braces have some more rules attached. A single letter or punctuation mark represents a control character (which comes out when you hold the control key (marked CTRL) down while pressing another key). For example, {C} is ctrl-C, and {.} is ctrl-period. A word is the name for a key, usually the name found on the key itself ({CLEAR} is the key labelled 'CLEAR'). A number says that the next key occurs that many times.

To clear the screen, and then go to the middle of the screen, our listings would say

{CLEAR 12 DOWN 20 RIGHT}

You would type the CLEAR key, the DOWN (ctrl-equals) key 12 times, and the RIGHT (ctrl-star) key 20 times.

A three-of-hearts would appear in our listings as

3{,}

You would type a 3, followed by a ctrl-comma. Since the 3 was outside of braces, it represents itself. The comma, on the other hand, is inside braces, and so represents a ctrl-comma, or a heart.

A square with a cross on it would be listed as

{Q W E DOWN 3 LEFT A S D DOWN 3 LEFT Z X C}

You would type ctrl-Q, ctrl-W, ctrl-E, the DOWN (ctrl-equals) key, the LEFT (ctrl-plus) key 3 times, ctrl-A, ctrl-S, ctrl-D, the DOWN key, the LEFT key 3 times, ctrl-Z, ctrl-X, and ctrl-C.

Atari:	Us:	You type:
	{A}	ctrl-A
	{B}	ctrl-B
	{C}	ctrl-C
	{D}	ctrl-D
	{E}	ctrl-E
	{F}	ctrl-F
	{G}	ctrl-G
	{H}	ctrl-H
	{I}	ctrl-I
	{J}	ctrl-J
	{K}	ctrl-K
	{L}	ctrl-L
	{M}	ctrl-M
	{N}	ctrl-N
	{O}	ctrl-O
	{P}	ctrl-P
	{Q}	ctrl-Q
	{R}	ctrl-R
	{S}	ctrl-S
	{T}	ctrl-T
	{U}	ctrl-U
	{V}	ctrl-V
	{W}	ctrl-W
	{X}	ctrl-X
	{Y}	ctrl-Y
	{Z}	ctrl-Z

Atari:	Us:	You type:
	{,}	ctrl-comma
	{.}	ctrl-period
	{;}	ctrl-semicolon
	{BACK}	ESC BACK
	{BELL}	ESC ctrl-2
	{CLEAR}	ESC shift-less or ESC ctrl-less
	{CLR TAB}	ESC ctrl-TAB
	{DEL CHAR}	ESC ctrl-BACK
	{DEL LINE}	ESC shift-BACK
	{DOWN}	ESC ctrl-equals
	{ESC}	ESC ESC
	{INS CHAR}	ESC ctrl-greater
	{INS LINE}	ESC shift-greater
	{LEFT}	ESC ctrl-plus
	{RIGHT}	ESC ctrl-star
	{SET TAB}	ESC shift-TAB
	{TAB}	ESC TAB
	{UP}	ESC ctrl-minus

---- Note ----

comma is ,
equals is =
greater is >
less is <
minus is -
period is .
plus is +
semicolon is ;
star is *

How Atari characters appear in Iridis listings.